[Previous Section | Next Section | Chapter Index | Main Index]

Section 5.2

Constructors and Object Initialization

Subsections

OBJECT TYPES IN JAVA are very different from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly constructed. For the computer, the process of constructing an object means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

5.2.1 Initializing Instance Variables

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named *PairOfDice*. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {

    public int die1 = 3;   // Number showing on the first die.
    public int die2 = 4;   // Number showing on the second die.

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The instance variables `die1` and `die2` are initialized to the values 3 and 4 respectively. These initializations are executed whenever a *PairOfDice* object is constructed. It's important to understand when and how this happens. There can be many *PairOfDice* objects. Each time one is created, it gets its own instance variables, and the assignments "`die1 = 3`" and "`die2 = 4`" are executed to fill in the values of those variables. To make this clearer, consider a variation of the *PairOfDice* class:

```
public class PairOfDice {

    public int die1 = (int)(Math.random()*6) + 1;
    public int die2 = (int)(Math.random()*6) + 1;

    public void roll() {
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Here, every time a new *PairOfDice* is created, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of **static** member variables, of course, the situation is quite different. There is only one copy of a `static` variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (int, double, etc.) are automatically initialized to zero if you provide no other values; boolean variables are initialized to `false`; and char variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is `null`. (In particular, since `Strings` are objects, the default initial value for *String* variables is `null`.)

5.2.2 Constructors

Objects are created with the operator, `new`. For example, a program that wants to use a *PairOfDice* object could say:

```
PairOfDice dice;   // Declare a variable of type PairOfDice.

dice = new PairOfDice();  // Construct a new object and store a
                          //   reference to it in the variable.
```

In this example, "`new PairOfDice()`" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, `dice`, so that after the assignment statement is executed, `dice` refers to the newly created object. Part of this expression, "`PairOfDice()`", looks like a subroutine call, and that is no accident. It is, in fact, a call to a special type of subroutine called a constructor. This might puzzle you, since there is no such subroutine in the class definition. However, every class has at least one constructor. If the programmer doesn't write a constructor definition in a class, then the system will provide a default constructor for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other subroutine, with three exceptions. A constructor does not have any return type (not even `void`). The name of the constructor must be the same as the name of the class in which it is defined. And the only modifiers that can be used on a constructor definition are the access modifiers `public`, `private`, and `protected`. (In particular, a constructor can't be declared `static`.)

However, a constructor does have a subroutine body of the usual form, a block of statements. There are no restrictions on what statements can be used. And a constructor can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the *PairOfDice* class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice(int val1, int val2) {
```

```
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
         die1 = val1;  // Assign specified values
         die2 = val2;  //           to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
         die1 = (int)(Math.random()*6) + 1;
         die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

The constructor is declared as "`public PairOfDice(int val1, int val2) ...`", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "`new PairOfDice(3,4)`" would create a *PairOfDice* object in which the values of the instance variables `die1` and `die2` are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;            // Declare a variable of type PairOfDice.

dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                            //   object that initially shows 1, 1.
```

Now that we've added a constructor to the *PairOfDice* class, we can no longer create an object by saying "`new PairOfDice()`"! The system provides a default constructor for a class **only** if the class definition does not already include a constructor. In this version of *PairOfDice*, there is only one constructor in the class, and it requires two actual parameters. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the *PairOfDice* class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice() {
            // Constructor.  Rolls the dice, so that they initially
            // show some random values.
        roll();  // Call the roll() method to roll the dice.
    }

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
        die1 = val1;  // Assign specified values
        die2 = val2;  //           to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Now we have the option of constructing a *PairOfDice* object either with "`new PairOfDice()`" or with "`new PairOfDice(x,y)`", where `x` and `y` are int-valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "`(int)(Math.random()*6)+1`", because it's done inside the *PairOfDice* class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the *PairOfDice* class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:

```
public class RollTwoPairs {

    public static void main(String[] args) {

        PairOfDice firstDice;  // Refers to the first pair of dice.
        firstDice = new PairOfDice();

        PairOfDice secondDice; // Refers to the second pair of dice.
        secondDice = new PairOfDice();

        int countRolls;  // Counts how many times the two pairs of
                         //    dice have been rolled.

        int total1;      // Total showing on first pair of dice.
        int total2;      // Total showing on second pair of dice.

        countRolls = 0;

        do {  // Roll the two pairs of dice until totals are the same.

            firstDice.roll();    // Roll the first pair of dice.
            total1 = firstDice.die1 + firstDice.die2;   // Get total.
            System.out.println("First pair comes up  " + total1);

            secondDice.roll();    // Roll the second pair of dice.
            total2 = secondDice.die1 + secondDice.die2;   // Get total.
            System.out.println("Second pair comes up " + total2);

            countRolls++;   // Count this roll.

            System.out.println();  // Blank line.

        } while (total1 != total2);

        System.out.println("It took " + countRolls
                           + " rolls until the totals were the same.");

    } // end main()

} // end class RollTwoPairs
```

Constructors are subroutines, but they are subroutines of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like `static` member subroutines, but they are not and cannot be declared to be `static`. In fact, according to the Java language specification, they are technically not members of the class at all! In particular, constructors are not referred to as "methods."

Unlike other subroutines, a constructor can only be called using the `new` operator, in an expression that has the form

```
new class-name ( parameter-list )
```

where the **parameter-list** is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a subroutine call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.
3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object.

For another example, let's rewrite the *Student* class that was used in Section 1. I'll add a constructor, and I'll also take the opportunity to make the instance variable, `name`, private.

```
public class Student {

   private String name;                 // Student's name.
   public double test1, test2, test3;   // Grades on three tests.

   public Student(String theName) {
        // Constructor for Student objects;
        // provides a name for the Student.
        // The name can't be null.
      if ( theName == null )
          throw new IllegalArgumentException("name can't be null");
      name = theName;
   }

   public String getName() {
        // Getter method for reading the value of the private
        // instance variable, name.
      return name;
   }

   public double getAverage() {
        // Compute average test grade.
      return (test1 + test2 + test3) / 3;
   }

}  // end of class Student
```

An object of type *Student* contains information about some particular student. The constructor in this class has a parameter of type *String*, which specifies the name of that student. Objects of type *Student* can be created with statements such as:

```
std = new Student("John Smith");
std1 = new Student("Mary Jones");
```

In the original version of this class, the value of `name` had to be assigned by a program after it created the object of type *Student*. There was no guarantee that the programmer would always remember to set the `name` properly. In the new version of the class, there is no way to create a *Student* object except by calling the constructor, and that constructor automatically sets the `name`. Furthermore, the constructor makes it impossible to have a student object whose name is `null`. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the `private` modifier. Since the instance variable, `name`, is `private`, there is no way for any part of the program outside the *Student* class to get at the `name` directly. The program sets the value of `name`, indirectly, when it calls the constructor. I've provided a getter function, `getName()`, that can be used from outside the class to find out the `name` of the student. But I haven't provided any setter method or other way to change the name. Once a student object is created, it keeps the same name as long as it exists.

Note that it would be legal, and good style, to declare the variable `name` to be "`final`" in this class. An instance variable can be `final` provided it is either assigned a value in its declaration or is assigned a value in every constructor in the class. It is illegal to assign a value to a `final` instance variable, except inside a constructor.

Let's take this example a little farther to illustrate one more aspect of classes: What happens when you mix static and non-static in the same class? In that case, it's legal for an instance method in the class to use static member variables or call static member subroutines. An object knows what class it belongs to, and it can refer to static members of that class. But there it only one copy of the static member, in the class itself. Effectively, all the objects share one copy of the static member.

As an example, consider a version of the *Student* class to which I've added an `ID` for each student and a `static` member called `nextUniqueID`. Although there is an `ID` variable in each student object, there is only one `nextUniqueID` variable.

```
public class Student {

   private String name;                 // Student's name.
   public double test1, test2, test3;   // Grades on three tests.

   private int ID;  // Unique ID number for this student.

   private static int nextUniqueID = 0;
             // keep track of next available unique ID number

   Student(String theName) {
        // Constructor for Student objects; provides a name for the Student,
        // and assigns the student a unique ID number.
      name = theName;
      nextUniqueID++;
      ID = nextUniqueID;
   }

   public String getName() {
        // Getter method for reading the value of the private
        // instance variable, name.
      return name;
   }

   public int getID() {
        // Getter method for reading the value of ID.
      return ID;
   }

   public double getAverage() {
```

```
            // Compute average test grade.
         return (test1 + test2 + test3) / 3;
      }

   }  // end of class Student
```

Since `nextUniqueID` is a `static` variable, the initialization "`nextUniqueID = 0`" is done only once, when the class is first loaded. Whenever a *Student* object is constructed and the constructor says "`nextUniqueID++;`", it's always the same static member variable that is being incremented. When the very first *Student* object is created, `nextUniqueID` becomes 1. When the second object is created, `nextUniqueID` becomes 2. After the third object, it becomes 3. And so on. The constructor stores the new value of `nextUniqueID` in the `ID` variable of the object that is being created. Of course, `ID` is an instance variable, so every object has its own individual `ID` variable. The class is constructed so that each student will automatically get a different value for its `ID` variable. Furthermore, the `ID` variable is `private`, so there is no way for this variable to be tampered with after the object has been created. You are guaranteed, just by the way the class is designed, that every student object will have its own permanent, unique identification number. Which is kind of cool if you think about it.

(Unfortunately, if you think about it a bit more, it turns out that the guarantee isn't quite absolute. The guarantee is valid in programs that use a single thread. But, as a preview of the difficulties of parallel programming, I'll note that in multi-threaded programs, where several things can be going on at the same time, things can get a bit strange. In a multi-threaded program, it is possible that two threads are creating *Student* objects at exactly the same time, and it becomes possible for both objects to get the same ID number. We'll come back to this in Subsection 12.1.3, where you will learn how to fix the problem.)

5.2.3 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically.

An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this in consecutive statements!):

```
Student std = new Student("John Smith");
std = null;
```

In the first line, a reference to a newly created *Student* object is stored in the variable `std`. But in the next line, the value of `std` is changed, and the reference to the *Student* object is gone. In fact, there are now no references whatsoever to that object, in any variable. So there is no way for the program ever to use the object again! It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." In the above example, it was very easy to see that the *Student* object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error, and it leads to problems when the

program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

[Previous Section | Next Section | Chapter Index | Main Index]